Marten - Import CPX Example

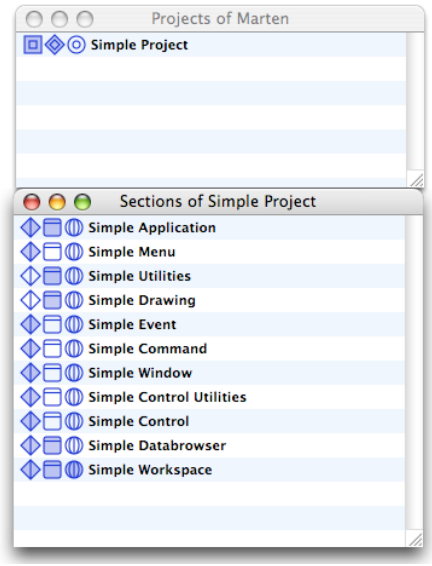
This document will show you how to import a Pictorius CPX section file into a Marten project. You should be familiar with the other Marten examples before starting this tutorial. You should also have downloaded the "Data File" CPX section file from the support page of the Andescotia website (www.andescotia.com/support/) or otherwise have access to it, perhaps from an existing directory of CPX files.

Important - Read Me

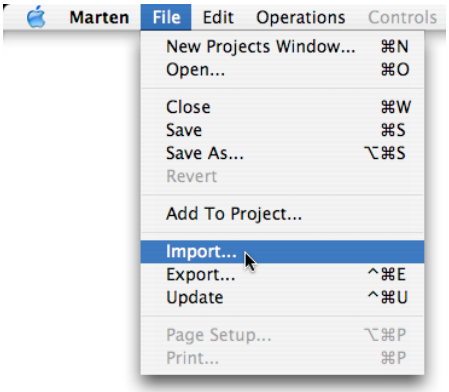
Because the Import functionality was developed by experimentation rather than from a detailed specification of the Prograph CPX section format, this functionality is to be considered experimental and Andescotia LLC makes no warranty regarding it. You use this functionality at your own risk, which certainly includes the risk of crashing the Marten IDE and loss of data. If you do not wish to take these risks, do not use the Import command. It is virtually certain that with Marten version 1.0, importing a large number of CPX files will lead to a crash sooner or later.

Basic Import

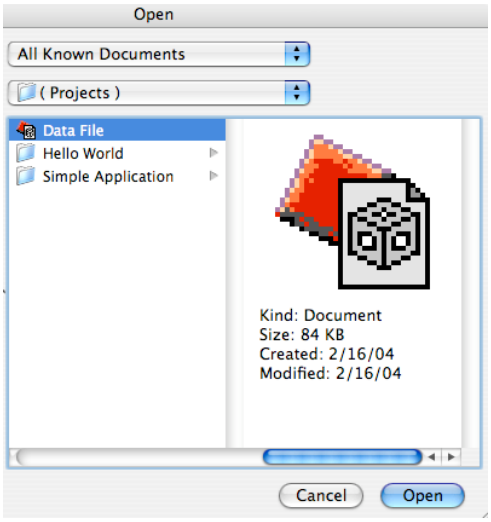
To begin, launch the Marten IDE and open the Simple Application project. Double-click the sections icon of the project item to open the Sections window.



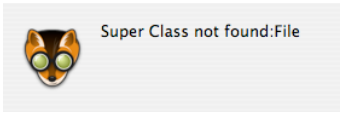
Select the Import command from under the File menu.



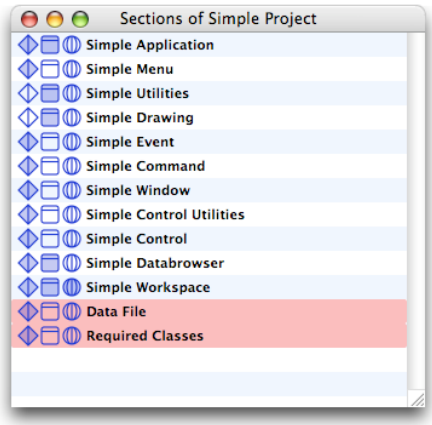
An Open File dialog window will open. Navigate to the Data File CPX section file, select it, and click the Open button. Important! Because of file name restrictions in Marten 1.0, a CPX file should have just low-ASCII characters in its name. Several CPX files have a bullet or other high-ASCII characters in their names. These files must be renamed before importing, otherwise the Marten 1.0 IDE will likely crash.



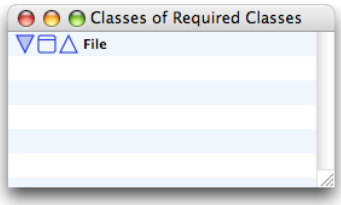
The Marten IDE will import the section file. You will see one warning dialog that states that when the section was added to the project, the parent class of Data File was not found. This parent class is the File class. Click the OK button to close the warning.



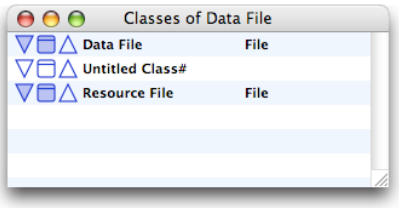
Examining the Sections window shows that not only was the Data File section imported and added to the project, but a new section was created called "Required Classes".



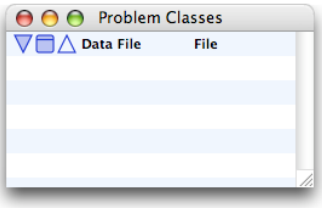
This section contains a newly created version of the File class so as to satisfy the parent class requirement of the Data File class.



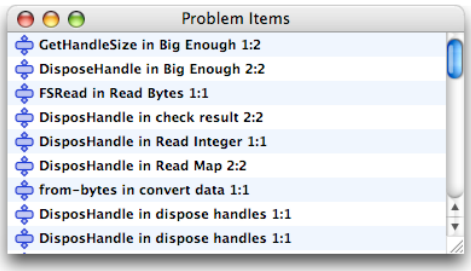
Examining the classes of the Data File section reveals that a new "Untitled Class#" was created. This class or one similar to it (with "Untitled Class" in the name) will always be created in an imported section if there were any problem items uncovered during the import process. This special class contains all the universal methods of the section as class methods of this special class. In this particular example, there actually are no universal methods of the Data File section. This special class will be discussed further later in the article.



There are two new windows that are open. The first window is entitled "Problem Classes" and lists the classes that had no defined parent class when the new section was added to the project.



We have already discussed the fact that the Data File class requires a File super class which was missing from the project when the section file was added. As a result, a File class was created and placed in the Required Classes section. This window is no longer needed, so close it and turn your attention to the second new window, the Problem Items window.

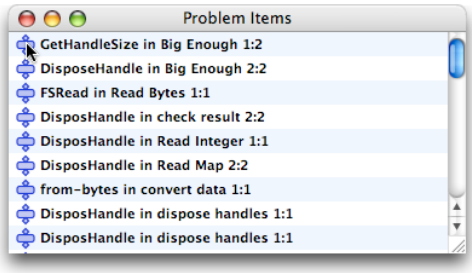


This window lists all the problem operations that were encountered during the import process. Do not close this window until you are certain that you will not need it any more because it cannot be reopened. The vast majority of problem operations are external procedures that either do not exist in Carbon or have an outarity (number of roots) mismatch. There is also a problem with an undefined primitive in the list shown above.

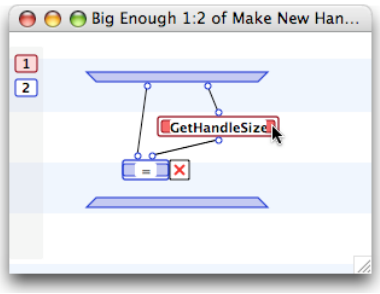
The next section illustrates how to deal with some of these problems.

Example Modifications

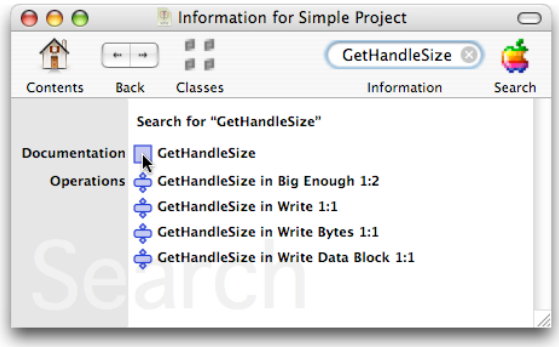
Double-click the first item in the list.



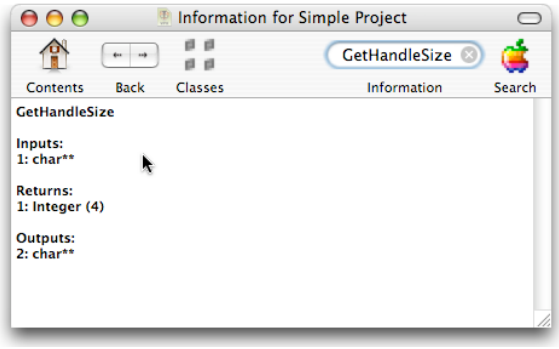
The case window containing this problem operation will open with the operation selected. Double-click the operation.



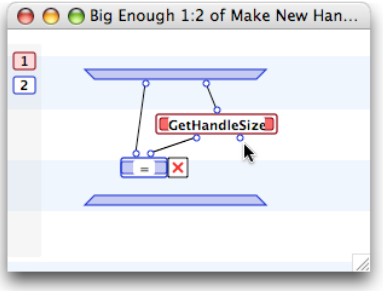
The Information Window for the project will open with the procedure name active. Double-click on the Documentation icon for the procedure to get the arity and type information for the external procedure.



The Information Window will now display the parameter information for the procedure.

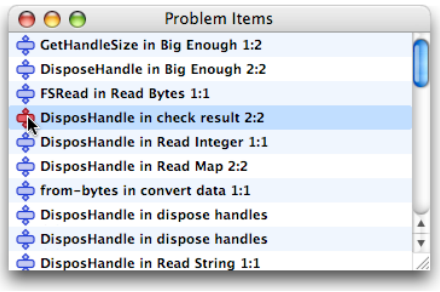


The problem was caused by the fact that the Marten tools require GetHandleSize to have two roots. The first root is needed to handle the function return of a 4 byte integer. The second root is required to handle the output of a character handle. Command-click just under the procedure to add a new root to the right of the original root.

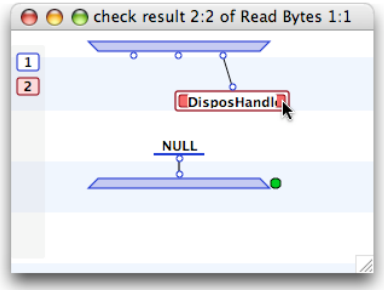


This fixes this particular problem. The mismatch of outarity between Prograph CPX and Marten is the number one source of problems during import, but fortunately is one of the easiest to fix.

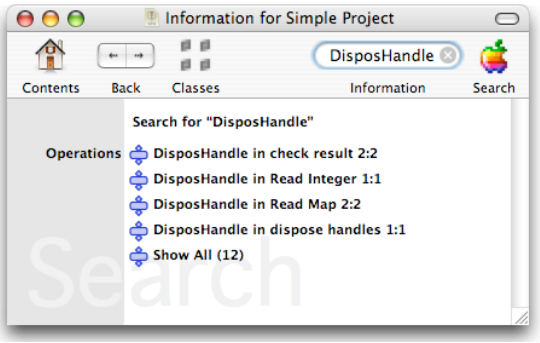
For your next example, double-click the fourth problem item in the Problem Items window.



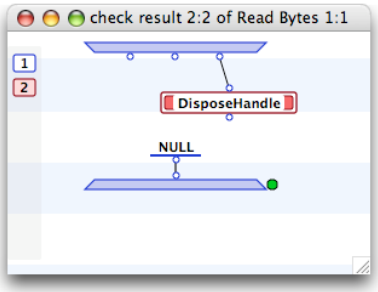
The case window containing this problem operation will open with the operation selected. Double-click the operation.



The Information Window for the project will open with the procedure name active. Notice that there is no documentation available! This is because this particular external procedure is not defined for Carbon. Fortunately, this procedure is probably an old one with a deprecated MacOS toolbox name which had letters missing to comply with the early toolbox naming restrictions.



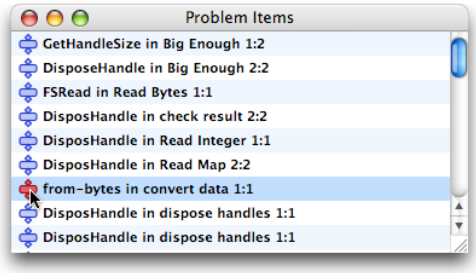
Try changing the name of the procedure to "DisposeHandle. The operation changes outarity indicating that it was looked up successfully and that this external procedure is defined in the Carbon library.



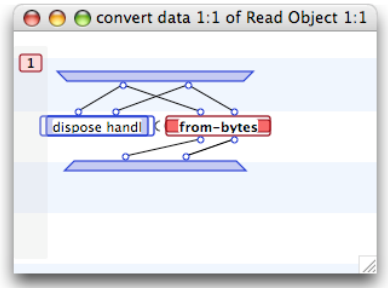
This problem has now been fixed. This type of problem is not as common as the arity mismatch one, but occurs regularly and is generally straightforward to fix. Occasionally the offending toolbox routine has simply been removed from the Carbon library and has no counterpart. For example, the Scrap Manager routines have been removed and new Carbon routines created. In that case, you will have to research the

new Carbon routines and modify the source to use them.

As the last example, double-click the "from-bytes" problem item, the seventh item in the Problem Items window.



Again, the case window containing the from-bytes primitive will open with the operation selected..



Double-clicking the operation will open the Information Window and as with the previous example, there will be no documentation available. This means that there is no primitive called "from-bytes". While the previous example could be fixed because an alternative name existed, there is no possibility of fixing this operation. Marten does not supply any primitives that can be used to extract data from the Pictorius CPX object format.

It should be understood that there will be Pictorius CPX primitives that do not have Marten counterparts. In situations like this, the imported code must be substantially modified or abandoned.

Unsupported CPX operations and further discussion

As you can see from these examples, that there can be substantial need to modify imported CPX code in order for it to function in the Marten environment. Here are some particulars:

The Marten IDE does not support:

1) The "partition" control nor the "TRUE" and "FALSE" roots.
2) The External Address operation.
3) The External Global operation.
4) Any preservation of comments, window sizes, or window locations.

If a local operation involves the partition control, you must rewrite the local to create the TRUE and FALSE output lists using multiple cases.

If you have External Address operations , you must convert them to External Get operations. Prograph CPX support for external structures involves creating new implicit datatypes that wrap the external structures. This is NOT how the Marten IDE supports external structures. To support an external structure, the Marten IDE creates a new External Block datum. This datum will store the type of structure, the size of the structure, the level of indirection (which for the case of a structure will be 0), and a block of memory containing the structure. As a consequence of this type of support, External Address operations are no longer needed and may be replaced with External Get operations.

The new MacOS X Carbon library has eliminated the need for the External Global operation. All former External Globals now have an external procedure counterpart. If you have External Globals in your code you must research their replacement in the documentation on the Apple Developer website.

The Marten support for external procedures is different than that of Prograph CPX. In particular the number of outputs (outarity) for a Marten version of an external procedure will often disagree with the Prograph CPX version. The first example illustrated that for the external procedure "GetHandleSize". The reason lies with the different tools used to generate the data dictionary that defines the interfaces for external procedures. In contrast with the CPX tools which generate interface information based upon the typedef (which in this case is

"Handle"), the Marten tools generate interface information based upon the ultimate type (which in this case is a "pointer to a pointer to a char").

The Marten IDE and the Marten tools treat external procedures very conservatively. Because GetHandleSize contains a non-constant pointer parameter, the Marten tools provide for an output root for the procedure. In addition, the Marten external procedure interface does not perform much in the way of type conversion. For example, an external procedure might have an input type of "char *" but a Marten String cannot be passed to such a terminal. You must use the primitive "string-to-pointer" to create an External Block datum which can be used instead. Similarly, an external procedure may have a root that supplies a pointer to an integer (for example, a "long *"). Marten will not create an Integer as an output, but instead will generate an External Block datum. You must use the primitive "get-integer" to extract the integer from the External Block.

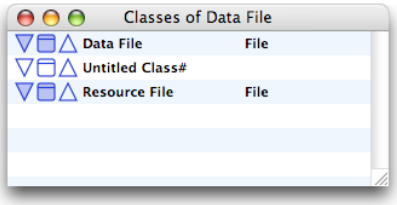
Marten does not have additional implicit datatypes such as Rect, Point, and RGB. Prograph CPX does and consequently these must be translated into a Marten datatype, which always the List datatype. The Prograph CPX data { 20 40 } (Point), { 65535 65535 65535 } (RGB), and { 20 40 120 240 } (Rect), will be translated into (20 40), (65535 65535 65535), and (20 40 120 240) Marten List data. To manipulate such lists, Marten supplies the "list-to-Point", "list-to-RGB", list-to-Rect", "Point-to-list", "RGB-to-list", and "Rect-to-list" primitives. It is important to understand that Marten does not insist on a representation of a Point as a list. You are free to represent these quantities as you wish, perhaps as an instance of a MyPoint class for example. It is just that when importing a CPX file, this is how these datatypes will be translated.

Universal Methods

Prograph CPX section files do not store universal methods as methods of the section. Instead a special class, which in Marten is named "Untitled Class" (perhaps with some #'s), is used to store such methods. In the rare occurrence when no problem items are found, Marten will move the methods of the Untitled Class to the universal methods of the section and delete the Untitled Class automatically. However, this is the exception and most of the time there will be problem items.

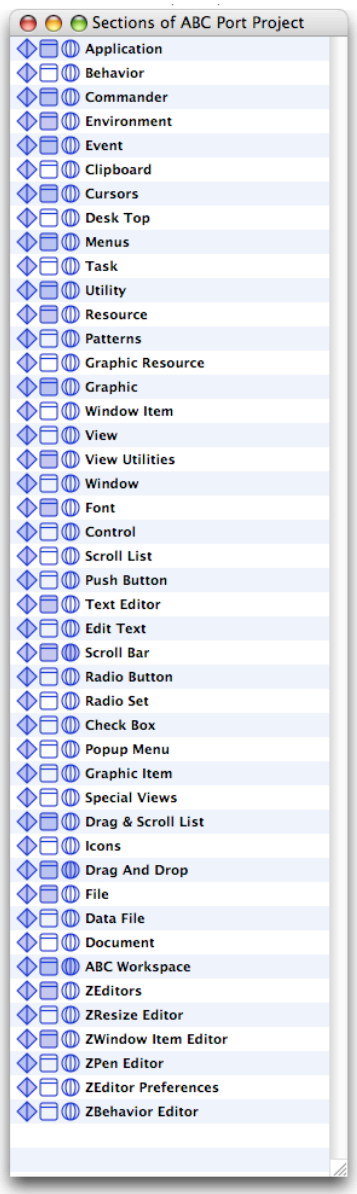
After you have performed as many modifications as you care to (you are not required to fix anything, but the problem item code will not execute successfully in the interpreter), close the Problem Items window and return to the Classes window of the imported section. In this case, we see that there are no methods of the Untitled Class to be moved. If there were, you would have opened the Methods window of the Untitled Class, selected all the methods, copied them, opened the Universals window of the Data File section, and then finally pasted them all in.

After the universal methods have been dealt with, you should select the Untitled Class and delete it.

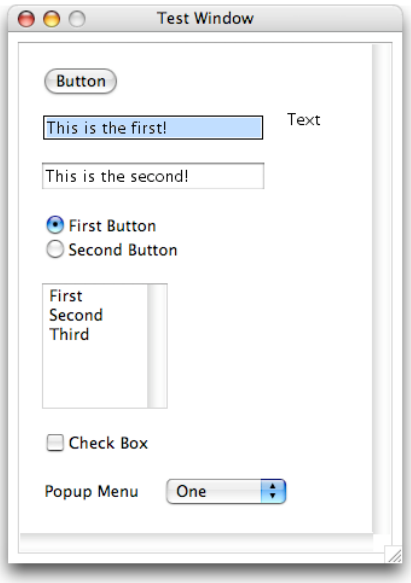


The ABCs and ABEs

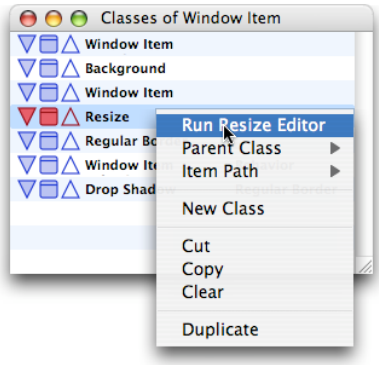
Andescotia does not supply an imported version of the Pictorius application framework known as the ABCs and ABEs. This is not to say that it could not be done. The screenshot below illustrates a partial port of the ABCs and ABEs created to support analysis of the import functionality and to support design of the Marten "editor" functionality.



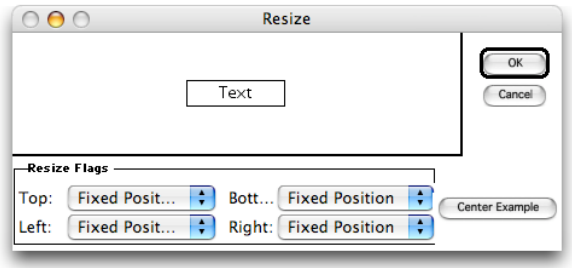
This porting project allowed a sample application to be written that opens a window with basic controls:



As stated, the porting project was undertaken in part to design the "editor" functionality. This functionality allows the user to create their own class and class instance editors. These editors are written in Prograph code and may be launched for a particular class by clicking on the class with the Control key held down. This will open up a contextual menu that will include a "Run ... Editor" command if an editor exists. An editor may be run on an instance of the class by selecting the attribute or persistent and hitting Command-E.



The next screenshot illustrates such an editor running.



Import strategy for the ABCs and ABEs

First of all, it is our recommendation that users do not attempt to import this set of CPX sections. As discussed above, there will be an almost insurmountable set of operations and code that will have to be modified to function correctly and it is not clear that all sections will even import without crashing, into a Marten project. However, the most important objection is that the application architecture of the ABCs is very outdated and unsupported. It is our very strong recommendation that anyone seeking to release a MacOS X version of an ABC application start from scratch and redesign it to use the latest MacOS X technologies, including Carbon Events, HI Controls and Views, and finally Core

Foundation.

That being said, we understand the desire of many to have this available. Andescotia will not undertake such a port, however the following describes a strategy that may help those who are determined to give it a try.

1) Rename all files that contain any high-ASCII characters in the file name. As discussed earlier, any ABE file that has a bullet in the file name must be renamed before importing.

2) Add sections one at a time, avoiding any undefined parent class problems. Do not try to import a group of sections all at once. The resulting problem items list could be too large and any non-importable sections could result in a crash that causes you to lose the ones that did import. You should research a strategy where you only import a section when all of the required parent classes have already been defined by previously imported sections. The list shown above illustrates such a strategy. First the Application class was imported because this class is not a subclass of any parent class. Next, the Behavior class was imported, then the Commander class, etc.

3) Fix problems as you can. When you import an ABC section, you will generally be presented with a list of problem operations. Fix as many as you comfortably can before closing the window, but don't worry about the rest. Although not present in Marten 1.0, code checking functionality will soon be available. Consequently, if you didn't get an operation fixed at import time, you will be able to return to it later.

4) Decide how you will represent Point, RGB, and Rect structures. Remember, these will be translated into lists initially but you may represent them however you like in your port. The simplest solution will be to leave them as lists and start modifying code using the "list-to-" and "-to-list" primitives.

5) Get something running as early as possible. The best way to fix many problems is to just deal with them as they come up as problems during execution. By the time you have imported the first 10 or so sections on the list, you will have enough code imported to start executing an application. Create a simple application in a workspace section and try it out. The interpreter will stop you soon enough at the first problem from the point of view of execution. Fix that problem and start execution again. Repeat this process until you get a basic application up and running. Then start adding in sections again.

6) Stick with it. Importing the ABCs will take a long time and will require a lot of work.